

# Automated Verification of Chapel Programs Using Model Checking and Symbolic Execution

**Timothy K. Zirkel   Stephen F. Siegel   Timothy McClory**

Verified Software Laboratory  
Department of Computer and Information Sciences  
University of Delaware

NFM 2013

## 1. Introduction

## 2. Challenges to using model checking

## 3. CIR: The Chapel Intermediate Representation

## 4. Example

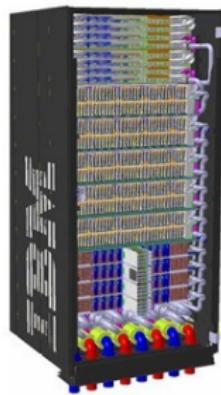
## 5. Experimental results

## 6. Future work

# New HPC Languages

## Examples

- Chapel
  - Cray
  - DARPA HPCS phase III
- X10
  - IBM
  - DARPA HPCS phase III
- Fortress
  - Sun Microsystems
  - DARPA HPCS phase II
- Titanium
  - UC Berkeley



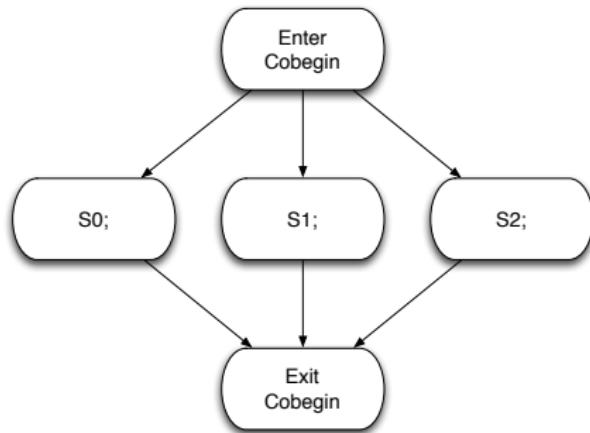
# Chapel themes

- General parallelism
- Global data view
- Locality control
- Data-centric synchronization
- Multiresolution
- Reduce gap between mainstream languages and HPC languages

# Cobegin

```
cobegin {  
    S1;  
    S2;  
    S3;  
}
```

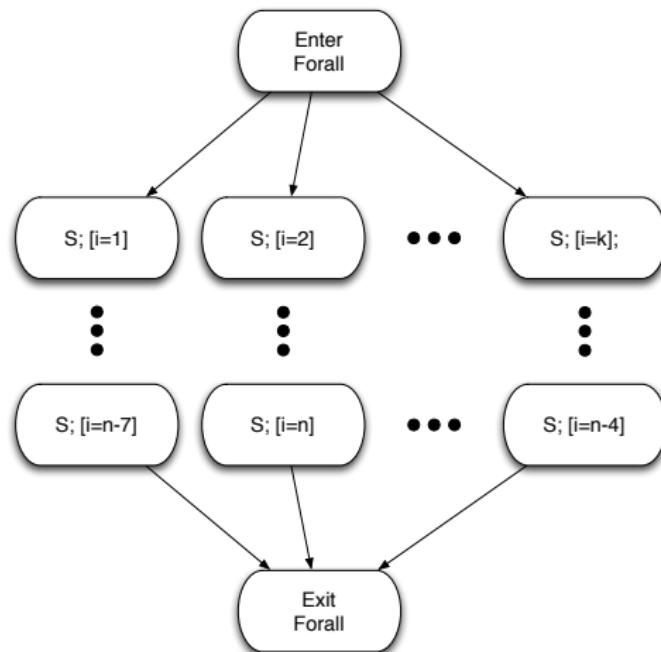
- A new task for each statement.
- At end, wait on all tasks.



# Forall

`forall i in <iteratorExpression> S`

- Iterations are executed by  $k$  tasks.
- $k$  is nondeterministic, could range from 1 to  $n$ .
- Partitioning of iterations among the  $k$  tasks is nondeterministic.
- **Coforall** to guarantee one task per iteration.



# Iterators

```
iter foo(arg0, ..., argN) : T {  
    ...;  
    yield e;  
    ...;  
}
```

- Yield statements instead of return.
- Persistent state.

## Sync variables

```
var sum: sync int;
```

- Have extra state: empty or full.
- Can only write when empty. A write changes the state to full.
- Can only read when full. A read changes the state to empty.
- Read/write and state change happens atomically.
- Allow data-centric synchronization.

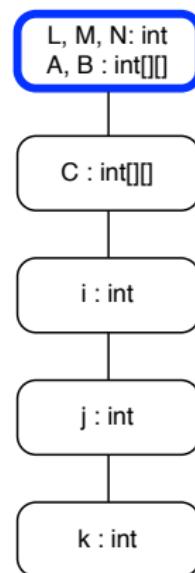
# Model checking challenges

```
config const L : int = 8;
config const M : int = 10;
config const N : int = 12;
var A : [0..L-1][0..M-1] int;
var B : [0..M-1][0..N-1] int;

proc main() {
    var C : [0..L-1][0..N-1] int;

    readInput();
    for i in 0..L {
        for j in 0..N {
            C[i][j] = 0;
            forall k in 0..M {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

## Static scopes



# Model checking challenges

```

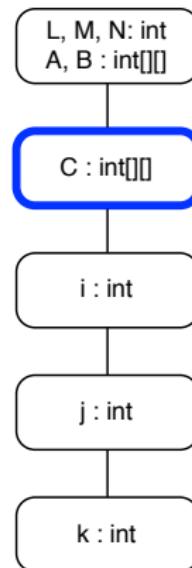
config const L : int = 8;
config const M : int = 10;
config const N : int = 12;
var A : [0..L-1][0..M-1] int;
var B : [0..M-1][0..N-1] int;

proc main() {
    var C : [0..L-1][0..N-1] int;

    readInput();
    for i in 0..L {
        for j in 0..N {
            C[i][j] = 0;
            forall k in 0..M {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}

```

## Static scopes



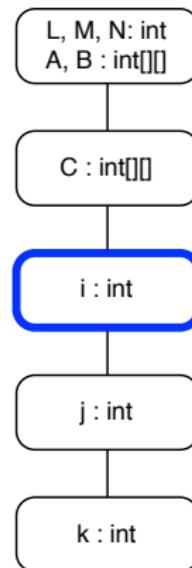
# Model checking challenges

```
config const L : int = 8;
config const M : int = 10;
config const N : int = 12;
var A : [0..L-1][0..M-1] int;
var B : [0..M-1][0..N-1] int;

proc main() {
    var C : [0..L-1][0..N-1] int;

    readInput();
    for i in 0..L {
        for j in 0..N {
            C[i][j] = 0;
            forall k in 0..M {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

## Static scopes



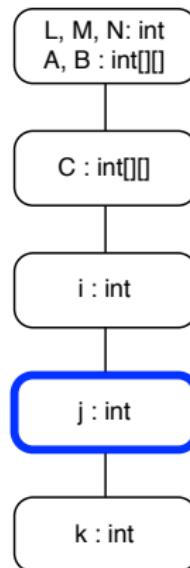
# Model checking challenges

```
config const L : int = 8;
config const M : int = 10;
config const N : int = 12;
var A : [0..L-1][0..M-1] int;
var B : [0..M-1][0..N-1] int;

proc main() {
    var C : [0..L-1][0..N-1] int;

    readInput();
    for i in 0..L {
        for j in 0..N {
            C[i][j] = 0;
            forall k in 0..M {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

## Static scopes



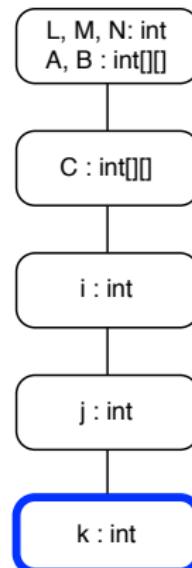
# Model checking challenges

```
config const L : int = 8;
config const M : int = 10;
config const N : int = 12;
var A : [0..L-1][0..M-1] int;
var B : [0..M-1][0..N-1] int;

proc main() {
    var C : [0..L-1][0..N-1] int;

    readInput();
    for i in 0..L {
        for j in 0..N {
            C[i][j] = 0;
            forall k in 0..M {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

## Static scopes



# Model checking challenges

```

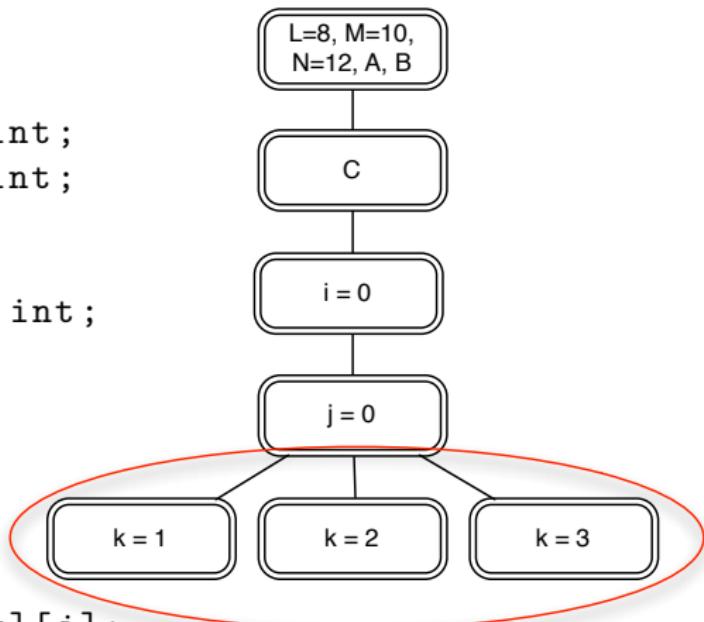
config const L : int = 8;
config const M : int = 10;
config const N : int = 12;
var A : [0..L-1] [0..M-1] int;
var B : [0..M-1] [0..N-1] int;

proc main() {
    var C : [0..L-1] [0..N-1] int;

    readInput();
    for i in 0..L {
        for j in 0..N {
            C[i][j] = 0;
            forall k in 0..M {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}

```

Dynamic scopes



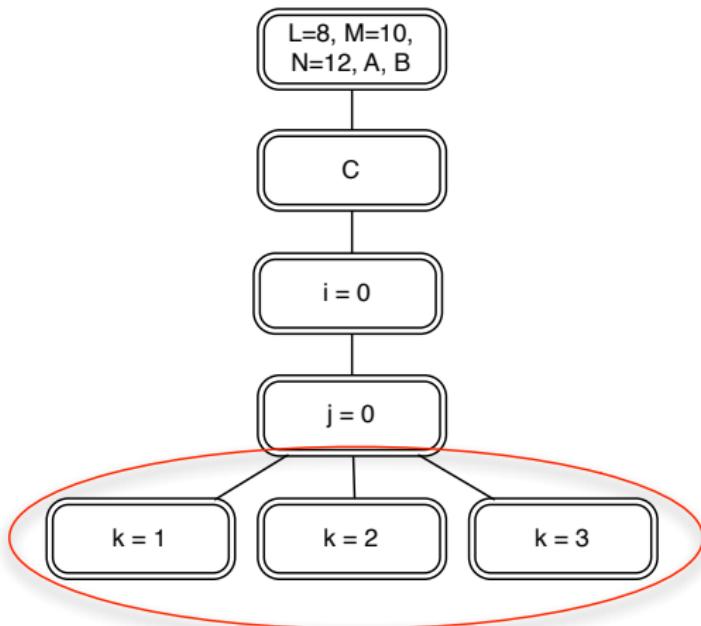
New tasks need to share non-root scopes!

# Model checking challenges

How to do this in, e.g., Promela?

- Language of model checker Spin.
- Dynamic process creation/destruction.
- Processes can only be declared in the global scope.

Dynamic scopes



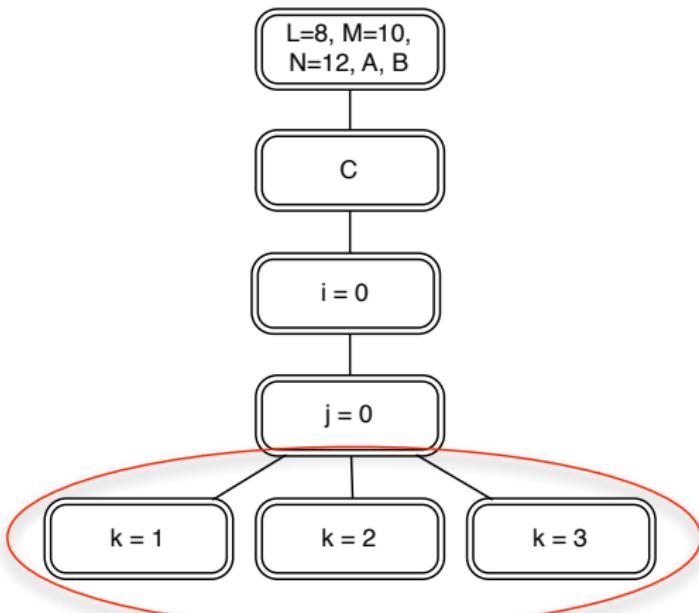
New tasks need to share non-root scopes!

# Model checking challenges

How to do this in, e.g., Promela?

- Define the process that would execute an iteration in the global scope.
- Put all shared variables in the global scope.
- Create multiple instances of each.
  - How many? Number of instantiations is dynamic.
- Might be possible, but not straightforward.
- Efficiency: lots of unused data in every state?
- Effectiveness of POR?

Dynamic scopes



New tasks need to share non-root scopes!

# CIR: The Chapel Intermediate Representation

- In-memory representation of a Chapel program.
- Scopes dynamically created and destroyed.
- Processes dynamically created and destroyed.

```
double Root(int n) { 0
    int i;
    double f1(double y) { 1
        if (y>0) { 2
            for (int i=0; i<n; i++) { 3
                ...
            }
        } else { 4
            double g(double z) { 5 ...
            } ...
        }
    }
    double f2(double w) { 6 ...
    } ...
}
```

# CIR model

Definition: a *CIR model* consists of

1. A set  $\Sigma$  of (static) scopes.
  - Has the structure of a rooted tree.
  - Root of the tree is  $\sigma_0$ .
2.  $\forall \sigma \in \Sigma$  a set of typed variables and a set of procedure symbols.
3. Types: *boolean*, *real*, *int*, *string*, *process*, arrays of any type
4. Procedures
  - Can be defined in any scope.

# CIR statements

1.  $v = e$
2. return  $e^*$
3.  $v^* = f(e_1, \dots, e_n)$
4.  $v^* = \text{fork } f(e_1, \dots, e_n)$ , where  $v$  has *process* type
5.  $\text{join}(e)$ , where  $e$  has *process* type
6.  $v = \text{choose}(e)$ , where  $v$  and  $e$  have integer type
7.  $\text{send}(\text{dest}, e, \text{tag})$ , where  $\text{dest}$  has *process* type,  $\text{tag}$  has integer type
8.  $\text{receive}(\text{src}, v, \text{tag})$  (like above, but  $\text{src}$  and  $\text{tag}$  may have form  $\text{any}(w)$ )
9.  $\text{write}(e)$
10.  $\text{noop}$
11.  $\text{sync-read}(b, v, x)$ , where  $b$  has boolean type
12.  $\text{sync-write}(b, x, e)$ , where  $b$  has boolean type.

\* indicates an optional component

## CIR semantics

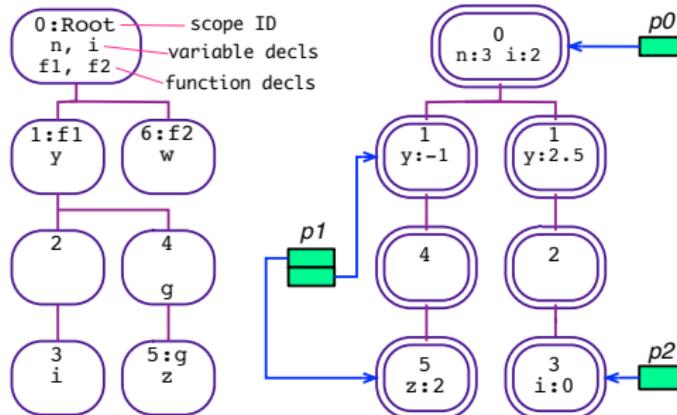
The semantics are specified in a small-step operational style using an interleaving view of concurrency.

The state of a CIR model comprises

1. a set  $\Delta$  of dynamic scopes;
2. values for each variable;
3. process IDs;
4. call stacks;
5. message buffers;

# CIR state

```
double Root(int n) { 0
    int i;
    double f1(double y) { 1
        if (y>0) { 2
            for (int i=0; i<n; i++) { 3
                ...
            }
        } else { 4
            double g(double z) { 5...
            } ...
        }
    }
    double f2(double w) { 6...
} ...
```

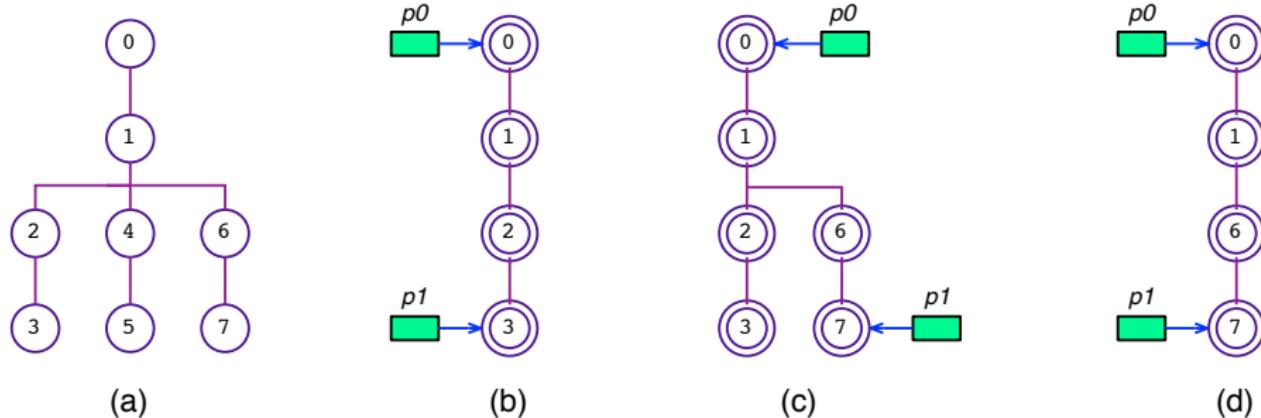


Pseudocode

Static scope tree

Example state

# Transition example: changing scopes



- a static scope tree;
- a dyscope tree;  $p_1$  is about to move from a location in scope 3 to a location in scope 7;
- new dyscopes are added corresponding to the path from scope 1 (the join of 3 and 7) to 7;
- dyscopes 2 and 3 became unreachable so were removed.

# Iterators

```
iter foo(arg0, ..., argN) : T {  
    ...; yield e; ...}
```

Chapel

CIR pseudocode

```
void foo(process caller, arg0, ..., argN) {  
    ...;  
    send(caller, e, _CVT_NEXT_TAG);  
    ...;  
    send(caller, NULL, _CVT_TERM_TAG);  
}
```

# Cobegin

`cobegin{S1; ...; SN}`

Chapel

---

CIR pseudocode

```
{ process _CVT_tmp_procs[N];
  void _CVT_tmp_1() {S1}
  ...
  void _CVT_tmp_N() {SN}
  _CVT_tmp_procs[0] = fork _CVT_tmp_1();
  ...
  _CVT_tmp_procs[N-1] = fork _CVT_tmp_N();
  join _CVT_tmp_procs[0];
  ...
  join _CVT_tmp_procs[N];
}
```

# Forall loops

```
forall x in f(...) S
```

- 
- Define worker procedure to execute body.
  - Nondeterministically choose number of worker tasks  $k$ .
  - Fork  $k$  workers.
  - Nondeterministically distribute iterations to workers.

# Example

```

var a : int;
var b: int;

void main() {
    var n : int = 10;

    cobegin{
        a = (n*(n+1))/2;
        for i in 0..n b+=i;
    }
}

int a, b; // Scope 0
void main() {
    int n = 10; // Scope 1
    process _CVT_tmp_procs[2];
    void _CVT_tmp_1() {a=(n*(n+1))/2; // Scope 2}
    void _CVT_tmp_2() { // Scope 3
        int i; process p; int tag;
        p = fork _CVT_range_iterator(self, 0, n);
        while (true) { // Scope 4
            receive(p, i, any(tag));
            if (tag == _CVT_TERM_TAG) break;
            b += i;
        }
        _CVT_tmp_procs[0] = fork _CVT_tmp_0();
        _CVT_tmp_procs[1] = fork _CVT_tmp_1();
        join _CVT_tmp_procs[0];
        join _CVT_tmp_procs[1];
    }
}

void _CVT_range_iterator(process caller, int lower,
                        int upper) {
    int current; // Scope 5
    current = lower;
    while (current <= upper) { // Scope 6
        send(caller, current, _CVT_NEXT_TAG);
        current = current + 1;
    }
    send(current, NULL, _CVT_TERM_TAG);
}

```

# Example

```

var a : int;
var b: int;

void main() {
    var n : int = 10;

    cobegin{
        a = (n*(n+1))/2;
        for i in 0..n b+=i;
    }
}

int a, b; // Scope 0
void main() {
    int n = 10; // Scope 1
    process _CVT_tmp_procs[2];
    void _CVT_tmp_1() {a=(n*(n+1))/2; // Scope 2}
    void _CVT_tmp_2() { // Scope 3
        int i; process p; int tag;
        p = fork _CVT_range_iterator(self, 0, n);
        while (true) { // Scope 4
            receive(p, i, any(tag));
            if (tag == _CVT_TERM_TAG) break;
            b += i;
        }
        _CVT_tmp_procs[0] = fork _CVT_tmp_0();
        _CVT_tmp_procs[1] = fork _CVT_tmp_1();
        join _CVT_tmp_procs[0];
        join _CVT_tmp_procs[1];
    }
}

void _CVT_range_iterator(process caller, int lower,
                        int upper) {
    int current; // Scope 5
    current = lower;
    while (current <= upper) { // Scope 6
        send(caller, current, _CVT_NEXT_TAG);
        current = current + 1;
    }
    send(current, NULL, _CVT_TERM_TAG);
}

```

# Example

```

var a : int;
var b: int;

void main() {
    var n : int = 10;

    cobegin{
        a = (n*(n+1))/2;
        for i in 0..n b+=i;
    }
}

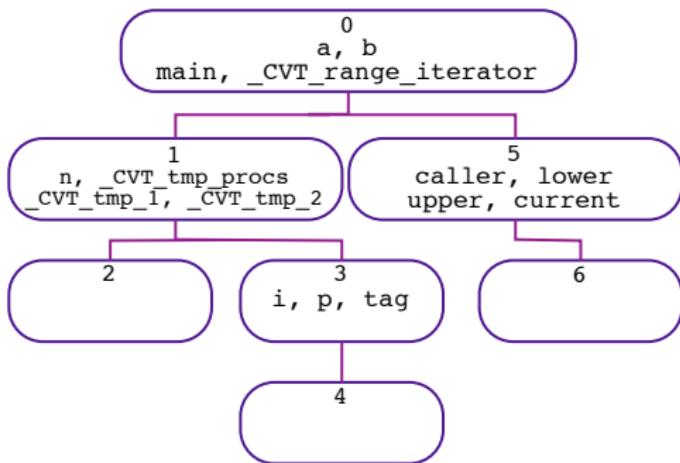
int a, b; // Scope 0
void main() {
    int n = 10; // Scope 1
    process _CVT_tmp_procs[2];
    void _CVT_tmp_1() {a=(n*(n+1))/2; // Scope 2}
    void _CVT_tmp_2() { // Scope 3
        int i; process p; int tag;
        p = fork _CVT_range_iterator(self, 0, n);
        while (true) { // Scope 4
            receive(p, i, any(tag));
            if (tag == _CVT_TERM_TAG) break;
            b += i;
        }
        _CVT_tmp_procs[0] = fork _CVT_tmp_0();
        _CVT_tmp_procs[1] = fork _CVT_tmp_1();
        join _CVT_tmp_procs[0];
        join _CVT_tmp_procs[1];
    }
}

void _CVT_range_iterator(process caller, int lower,
                        int upper) {
    int current; // Scope 5
    current = lower;
    while (current <= upper) { // Scope 6
        send(caller, current, _CVT_NEXT_TAG);
        current = current + 1;
    }
    send(current, NULL, _CVT_TERM_TAG);
}

```

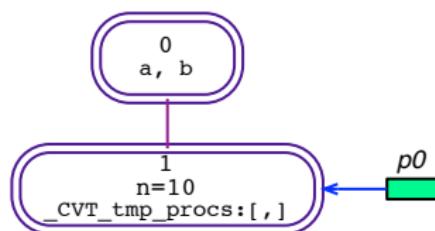
# Static scope tree

```
var a : int;  
var b: int;  
  
void main() {  
    var n : int = 10;  
  
    cobegin{  
        a = (n*(n+1))/2;  
        for i in 0..n b+=i;  
    }  
}
```



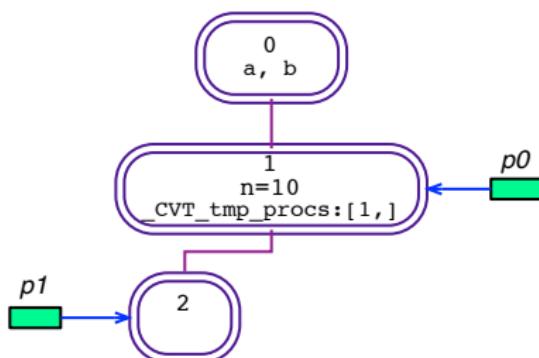
# State snapshot

```
var a : int;  
var b: int;  
  
void main() {  
    var n : int = 10;  
  
    cobegin{  
        a = (n*(n+1))/2;  
        for i in 0..n b+=i;  
    }  
}
```



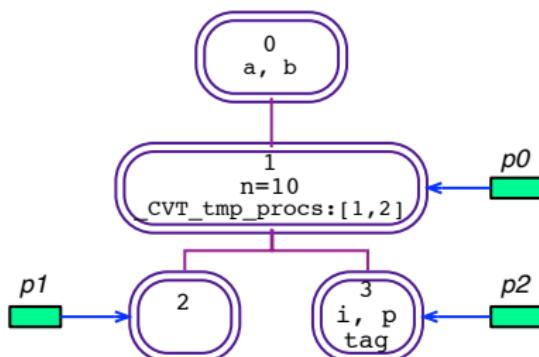
# State snapshot

```
var a : int;  
var b: int;  
  
void main() {  
    var n : int = 10;  
  
    cobegin{  
        a = (n*(n+1))/2;  
        for i in 0..n b+=i;  
    }  
}
```



# State snapshot

```
var a : int;  
var b: int;  
  
void main() {  
    var n : int = 10;  
  
    cobegin{  
        a = (n*(n+1))/2;  
        for i in 0..n b+=i;  
    }  
}
```



# CVT: The Chapel Verification Tool

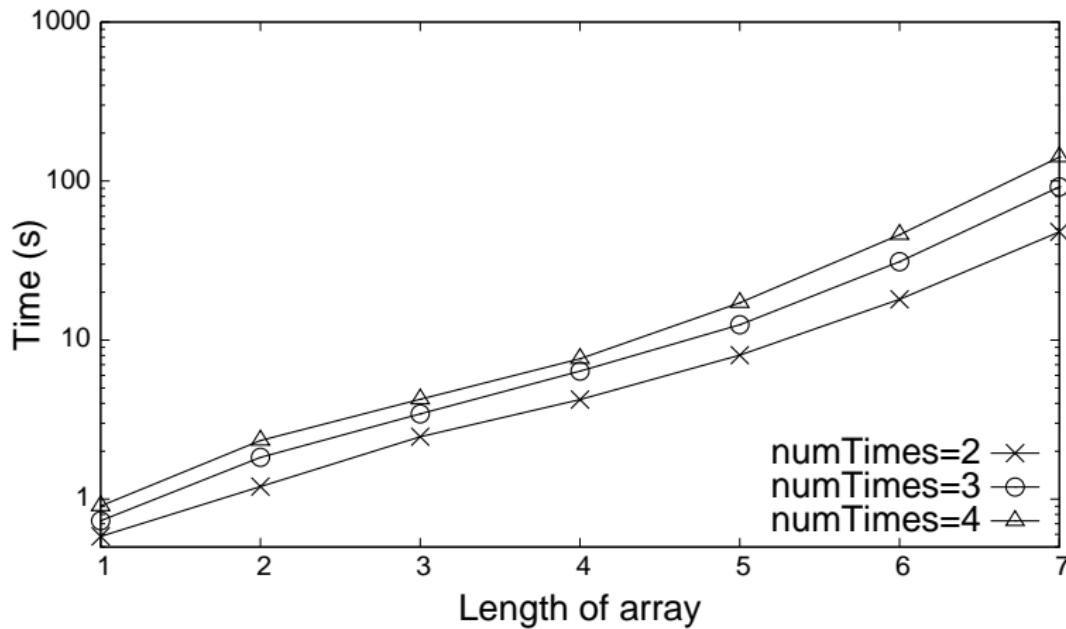
Parses a subset of Chapel.

```
prog ::= decl+
decl ::= (config | extern)? (const | var | param) v : type (= expr)? ;
       | (iter | proc) f ( (v : type , v : type)* )? (: type)? block
block ::= { (decl | stmt)* }
type ::= sync? (real | int | void | bool | string | [ expr ] type)
stmt ::= block | call | expr = expr ; | return expr? ; | yield expr ;
       | while ( expr ) block | if ( expr ) then stmt (else stmt)?
       | cobegin block
       | (for | forall | coforall) v in (expr .. expr | call) block
expr ::= x | int_literal | float_literal | string_literal | true | false
       | call | expr [ expr ] | ( expr ) | (+ | - | !) expr
       | expr (|| | && | == | != | < | > | >= | <= | + | - | * | / | %) expr
call ::= f ( (expr , expr)* )?
```

# CVT: The Chapel Verification Tool

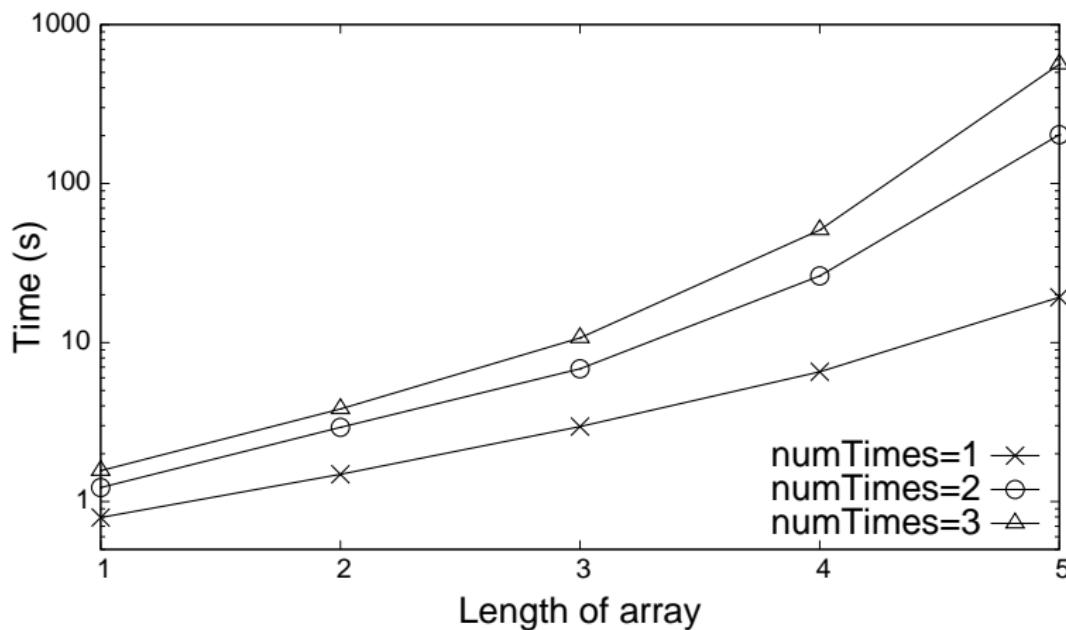
- Converts to a CIR model.
- Model checking with symbolic execution.
  - Small scope hypothesis.
- Verifies correctness properties of a single program. Absence of:
  - Deadlock.
  - Variable use before declaration.
  - Array index out of bounds.
  - Division by 0.
- Compare two programs for functional equivalence.
  - Comparative symbolic execution.
    - Siegel, Miranova, Avrunin, Clarke, TOSEM 2008  
Combining symbolic execution with model checking to verify parallel numerical programs.
    - Same technique used in TASS.
      - Siegel, Zirkel, Mathematics in Computer Science 2011  
TASS: The Toolkit for Accurate Scientific Software.
      - Useful for detecting data races, other nondeterministic behavior.
  - Tested on 11 synthetic Chapel programs.

## Verification of adderPar



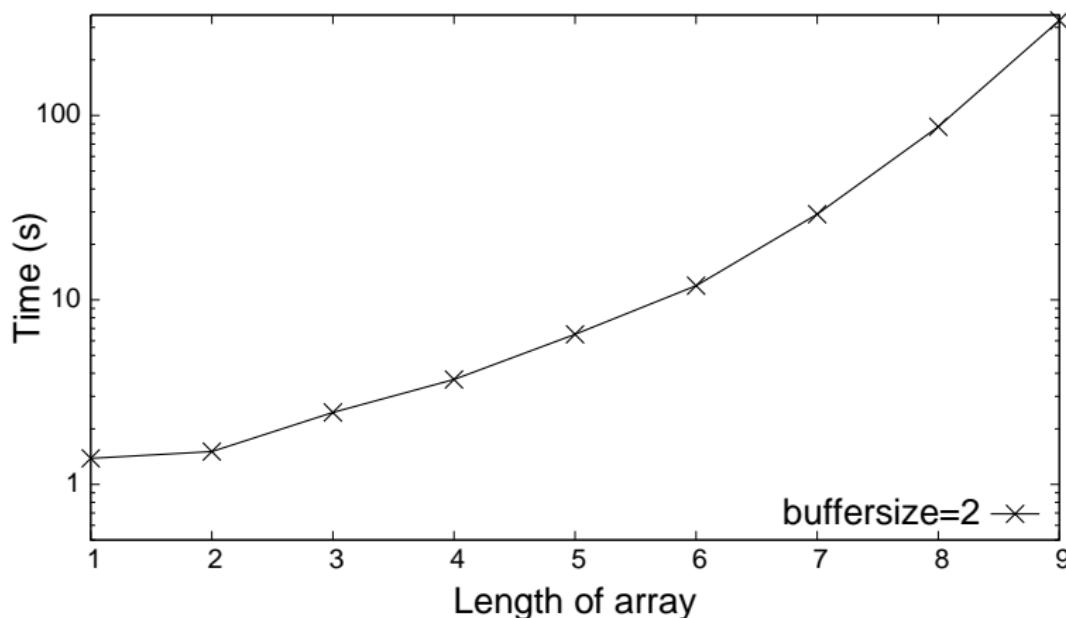
- Max # of workers per `forall` loop: 2
- Max # of concurrent processes seen: 9

# Checking functional equivalence of *adderSpec* and *adderPar*



- Max # of workers per `forall` loop: 3
- Max # of concurrent processes seen: 10

## Verification of *prodCons*



- Max # of concurrent processes seen: 5

# Detecting defects

Experiment	workers	states	transitions	procs	time (s)
<i>adderPar vs adderNoSync</i>	20	14495	14496	10	6.260
<i>adderSpec vs adderND</i>	2	7675	7819	12	5.027
<i>locks</i>	N/A	5562	5742	5	2.712
<i>cycle</i>	20	1067	1066	4	1.224
<i>prodCons vs prodConsNoSync</i>	N/A	3011	3010	6	2.779

## Future work

- Support additional Chapel features.
- Verify more complex examples.
- Improve performance.